新的步骤，首页篇。

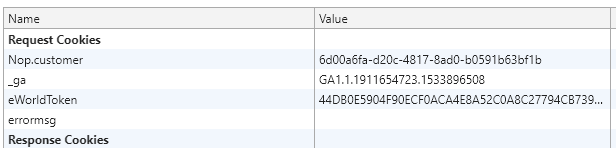
是

1.访问首页

（1）

访问该地址，

cookie发过去



后台处理，设置了errormsg的cookie

#region view[平台首页]

public ActionResult PlatformIndexPage()

{

if (HttpContext.Request.Cookies["errormsg"] == null)

{

HttpCookie mycookie = new HttpCookie("errormsg");

mycookie.Value = "";

mycookie.Expires = DateTime.Now.AddHours(1);

Response.Cookies.Add(mycookie);

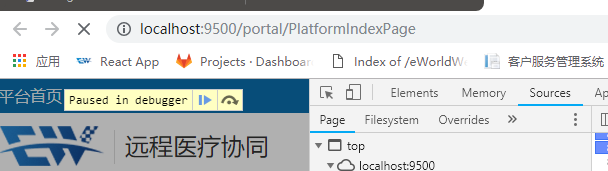
}

return View();

}

返回给前端当前视图：

前台设置了断点，所以一直菊花状态。



js调用两个方法：绑定诊断中心列表和绑定专家列表

<script type="text/javascript">

BindCenterList();

BindExpertList();

方法内部都是ajax请求，对应监控到的这两个请求。



其中绑定专家列表中内嵌了这个也是调用ajax的绑定图片的方法：

var imgUrl = ReturnImgUrl(CenterPicGuid, ["/Content/productStyle/@BaseCommon.ProductColorStyle/img/protall/serviceCenterD.jpg");](mailto:\"/Content/productStyle/@BaseCommon.ProductColorStyle/img/protall/serviceCenterD.jpg\");)

对应监视到的：



（1）其中绑定诊断中心列表：

地址对应的ca：

/// 首页获取推荐诊断中心

/// </summary>

/// <returns></returns>

public JsonResult GetCenterListHome()

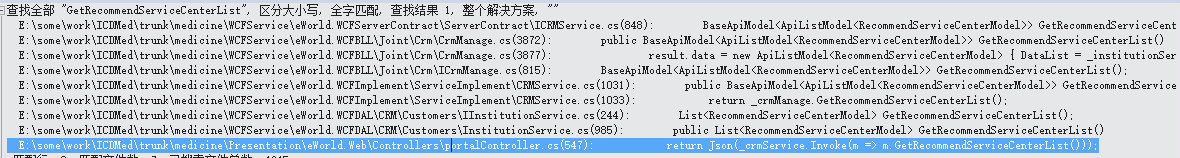
{

return Json(\_crmService.Invoke(m => m.GetRecommendServiceCenterList()));

}

搜索该方法，还是





上面的ca，也就是ui层的调用，接下来走到了服务端：

a.wcf服务端的契约

[OperationContract]

BaseApiModel<ApiListModel<RecommendServiceCenterModel>> GetRecommendServiceCenterList();

b.wcf服务端的实现：

实现调的是CrmManage（所有bll层Service对象的大集合，相当于所对应的，这里是crm，所有的bll层的方法都在里面）的实现。

namespace eWorld.WCFImplement.ServiceImplement

{

/// <summary>

/// 依赖注入属性 实例化该类从Ioc容器中获取

/// </summary>

[IOCServiceBehavior]

public class CRMService : ICRMService

{

#region Fields

//基础数据Service

private readonly ICrmManage \_crmManage;

private readonly ICommonManage \_iCommonMange;

/// 获取首页推荐服务中心列表

/// </summary>

/// <returns></returns>

public BaseApiModel<ApiListModel<RecommendServiceCenterModel>> GetRecommendServiceCenterList()

{

return \_crmManage.GetRecommendServiceCenterList();

}

其中这里bll层的实现，有ioc注入，我没有仔细去研究。



仔细一看确实是的，没有之前管理类那么多，复杂的readonly字段，以及构造函数。

c.管理类manage接口

public interface ICrmManage

{

/// 获取首页推荐服务中心列表

/// </summary>

/// <returns></returns>

BaseApiModel<ApiListModel<RecommendServiceCenterModel>> GetRecommendServiceCenterList();

d.管理类manager中的实现：

namespace eWorld.WCFBLL.Joint.Crm

{

public class CrmManage : ICrmManage

{

构造函数就是加入所有的bllService对象。

#region 全局变量

private readonly IInstitutionService \_institutionService;

private readonly IDoctorMemberService \_doctorMemberService;

……………………

#region 构造函数

public CrmManage(IInstitutionService institutionService,

IDoctorMemberService doctorMemberService,

………………………………

/// 获取首页推荐服务中心列表

/// </summary>

/// <returns></returns>

public BaseApiModel<ApiListModel<RecommendServiceCenterModel>> GetRecommendServiceCenterList()

{

var result = new BaseApiModel<ApiListModel<RecommendServiceCenterModel>> { code = ResultStatusCode.OK, msg = "成功" };

try

{

result.data = new ApiListModel<RecommendServiceCenterModel> { DataList = \_institutionService.GetRecommendServiceCenterList() };

}

catch (Exception ex)

{

SystemLogHelper.LogInsert(new AddLogParam { ShortMessage = "协同-[Crm]-获取首页推荐服务中心列表", FullMessage = $"详细错误：{ex}" });

result.code = ResultStatusCode.SYSTEM\_BUSY;

result.msg = "请求异常";

}

return result;

}

这里面引用的接口，再扩展。

TODO。

e.接下来就到了WCF的Dal层。显示DAL层的接口。

namespace eWorld.WCFDAL.CRM.Customers

{

public partial interface IInstitutionService

{

/// 获取首页推荐服务中心列表

/// </summary>

/// <returns></returns>

List<RecommendServiceCenterModel> GetRecommendServiceCenterList();

f.wcf dal层的实现：

这里和bll层的manage管理类很像，就相当于把dal层的所有对象在这里初始化。这里的一个个仓库字段，才是真正地介入到了ef中。调用了eWorld.Entity。

namespace eWorld.WCFDAL.CRM.Customers

{

public partial class InstitutionService : IInstitutionService

{

#region 私有对象

private readonly ICRMRepository<Institution> \_institutionRepository;

private readonly ICRMRepository<InstitutionManager> \_institutionManagerRepository;

private readonly ICRMRepository<ServiceCenter> \_serviceCenterRepository;

private readonly ICRMRepository<ServiceOpen> \_serviceOpenRepository;

private readonly ICRMRepository<DoctorServiceCenter> \_doctorServiceCenterRepository;

………………………………………………（省略很多私有对象）

构造函数中的内容也是多的一匹！

public InstitutionService(

ICRMRepository<Institution> institutionRepository,

ICRMRepository<InstitutionManager> institutionManagerRepository,

ICRMRepository<ServiceCenter> serviceCenterRepository,

ICRMRepository<DoctorServiceCenter> doctorServiceCenterRepository,

………………………………（省略了很多）

{

\_institutionRepository = institutionRepository;

\_institutionManagerRepository = institutionManagerRepository;

\_serviceCenterRepository = serviceCenterRepository;

\_doctorServiceCenterRepository = doctorServiceCenterRepository;

…………………………（省略很多）

/// 获取首页推荐服务中心列表

/// </summary>

/// <returns></returns>

public List<RecommendServiceCenterModel> GetRecommendServiceCenterList()

{

var query = from sc in \_serviceCenterRepository.Table

where sc.State == 10

&& sc.IsDisplay == true

&& sc.IsRecommend == true

orderby sc.CreateDate descending

select new

{

sc.Id,

sc.ServiceCenterName,

sc.HospitalName,

sc.CreateDate

};

var list = query.ToList();

var result = new List<RecommendServiceCenterModel>();

foreach (var item in list)

{

result.Add(new RecommendServiceCenterModel

{

ServiceCenterId = item.Id,

ServiceCenterName = item.ServiceCenterName,

HospitalName = item.HospitalName,

CreateDate = item.CreateDate.GetStringDateTimeValue(),

PicList = (from scp in \_serviceCenterPicRepository.Table where scp.ServiceCenterId == item.Id && scp.Kind == 20 select scp.PicGuid).ToList()

});

}

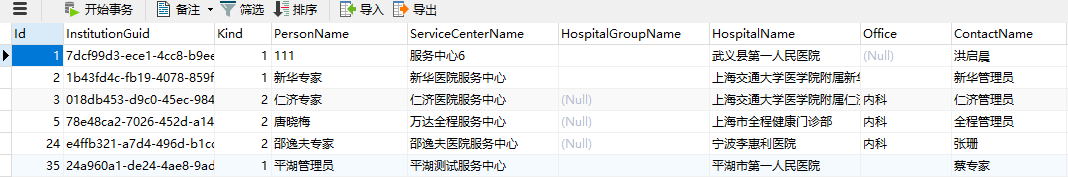
return result;

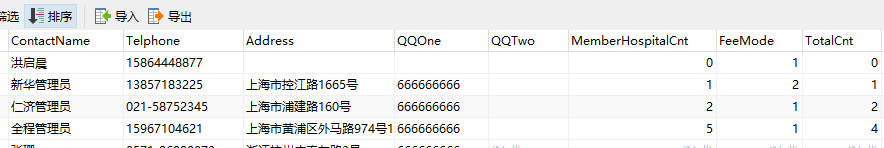
}

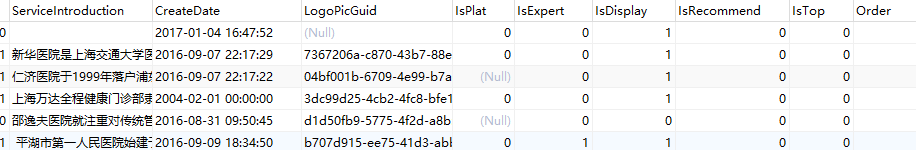
进一步查找，去数据库中看了。

CRM数据库的ServiceCenter找到了想要的内容：

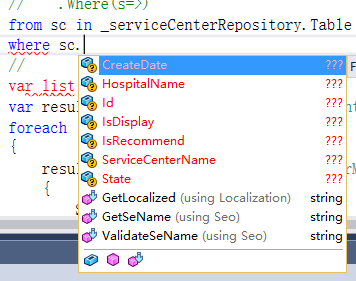






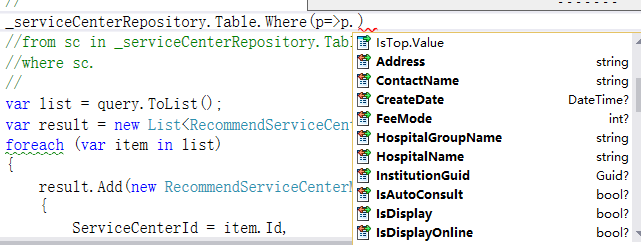


怎么用linq点出来了。



lambda点出来的是方法……

重新生成了下，没有错误了，点出来了！



再细究一下执行代码：

/// 获取首页推荐服务中心列表

/// </summary>

/// <returns></returns>

public List<RecommendServiceCenterModel> GetRecommendServiceCenterList()

{

从\_serviceCenter对应的实体，也就是ServiceCenter数据库中获取数据。

var query = from sc in \_serviceCenterRepository.Table

按条件获取

where sc.State == 10

&& sc.IsDisplay == true

&& sc.IsRecommend == true

orderby sc.CreateDate descending

最后选取自己想要的，因为数据库中的字段数量有二三十个，不可能都要的。

select new

{

sc.Id,

sc.ServiceCenterName,

sc.HospitalName,

sc.CreateDate

};

这里从Queryable变成Enumerable，就是从数据库中加载到了内存中。

var list = query.ToList();

转化为新的格式，这个格式是和前端约定好的。也可以说是接口规范。

var result = new List<RecommendServiceCenterModel>();

foreach (var item in list)

{

result.Add(new RecommendServiceCenterModel

{

ServiceCenterId = item.Id,

ServiceCenterName = item.ServiceCenterName,

HospitalName = item.HospitalName,

获取规定日期的时间

CreateDate = item.CreateDate.GetStringDateTimeValue(),

从诊断中心图片数据库中获取图片的地址。

PicList = (from scp in \_serviceCenterPicRepository.Table where scp.ServiceCenterId == item.Id && scp.Kind == 20 select scp.PicGuid).ToList()

});

}

return result;

}

日期转换的方法：

这里是静态扩展类：

namespace eWorld.Core

{

/// <summary>

/// 参数验证及获取控制公共类

/// </summary>

public static class ArgumentHelper

{

/// <param name="value"></param>

/// <param name="formatstr"></param>

/// <returns></returns>

public static string GetStringDateTimeValue(this object value, string formatstr = "yyyy-MM-dd")

{

if (value != null && !string.IsNullOrEmpty(value.ToString()))

{

return Convert.ToDateTime(value).ToString(formatstr);

}

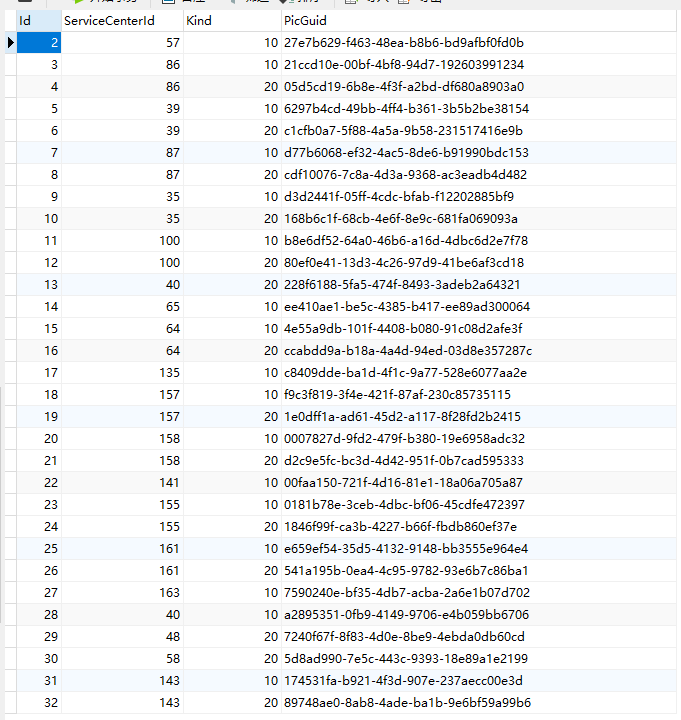
return string.Empty;

}

获取诊断中心对应的图片地址，再让前台去访问这个地址。数量不多，32个。想到了之前的海量图片模式，就要用文件服务器了。

PicList = (from scp in \_serviceCenterPicRepository.Table where scp.ServiceCenterId == item.Id && scp.Kind == 20 select scp.PicGuid).ToList()





返回的list<>中对应的数据类型：

在Core.DTO中

namespace eWorld.Core.DTO.portal

{

/// <summary>

/// 推荐服务中心实体类

/// </summary>

public class RecommendServiceCenterModel

{

/// <summary>

/// 服务中心id

/// </summary>

public int ServiceCenterId { get; set; }

/// <summary>

/// 服务中心名称

/// </summary>

public string ServiceCenterName { get; set; }

/// <summary>

/// 医院名称

/// </summary>

public string HospitalName { get; set; }

/// <summary>

/// 创建时间

/// </summary>

public string CreateDate { get; set; }

/// <summary>

/// 图片guid列表

/// </summary>

public List<Guid?> PicList { get; set; }

}

}

查找dto：

DTO就是前后端数据交互中用来承载数据的对象，这个对象被填充数据后就会被序列化为json、xml或其他格式，然后通过http等网络协议发送给对方。

查找2：

DTO即数据传输对象。之前不明白有些框架中为什么要专门定义DTO来绑定表现层中的数据，为什么不能直接用实体模型呢，有了DTO同时还要维护DTO与Model之间的映射关系，多麻烦。

然后看了这篇文章中的讨论部分才恍然大悟。

摘两个比较有意义的段落。

表现层与应用层之间是通过数据传输对象（DTO）进行交互的，数据传输对象是没有行为的POCO对象，它 的目的只是为了对领域对象进行数据封装，实现层与层之间的数据传递。为何不能直接将领域对象用于 数据传递？因为领域对象更注重领域，而DTO更注重数据。不仅如此，由于“富领域模型”的特点，这样 做会直接将领域对象的行为暴露给表现层。

需要了解的是，数据传输对象DTO本身并不是业务对象。数据传输对象是根据UI的需求进行设计的，而不 是根据领域对象进行设计的。比如，Customer领域对象可能会包含一些诸如FirstName, LastName, Email, Address等信息。但如果UI上不打算显示Address的信息，那么CustomerDTO中也无需包含这个 Address的数据

简单来说Model面向业务，我们是通过业务来定义Model的。而DTO是面向界面UI，是通过UI的需求来定义的。通过DTO我们实现了表现层与Model之间的解耦，表现层不引用Model，如果开发过程中我们的模型改变了，而界面没变，我们就只需要改Model而不需要去改表现层中的东西。

g.深入地去理解下项目中ef的踪影

\_serviceCenterPicRepository.Table

（1）

\_serviceCenterPicRepository对应的是

private readonly ICRMRepository<ServiceCenter> \_serviceCenterRepository;

继续去查找ICRMRepository<ServiceCenter>：

（2）

定义接口是：

这里已经走进了eWorld.Core.Data，正式进入到ef范畴了。

namespace eWorld.Core.Data

{

/// <summary>

/// Repository

/// </summary>

public partial interface ICRMRepository<T> where T : class

{

T GetById(object id);

void Insert(T entity);

void Update(T entity);

void Delete(T entity);

void DeleteById(object Id);

IQueryable<T> Table { get; }

}

}

（3）具体实现是  
namespace eWorld.Data

{

private readonly IDbContext \_context;

private IDbSet<T> \_entities;

/// Ctor

/// </summary>

/// <param name="context">Object context</param>

public CRMRepository(IDbContext context)

{

this.\_context = context;

}

/// <summary>

/// Entity Framework repository

/// </summary>

public partial class CRMRepository<T> : ICRMRepository<T> where T : BaseEntity

{

public virtual IQueryable<T> Table

{

get

{

return this.Entities;

}

}

protected virtual IDbSet<T> Entities

{

get

{

if (\_entities == null)

\_entities = \_context.Set<T>();

return \_entities;

}

}

进一步查找BaseEntity：

到了eWorld.Entity中。

namespace eWorld.Entity

{

/// <summary>

/// Base class for entities

/// </summary>

[Serializable]

public abstract partial class BaseEntity

{

/// <summary>

/// Gets or sets the entity identifier

/// </summary>

public int Id { get; set; }

public override bool Equals(object obj)

{

return Equals(obj as BaseEntity);

}

private static bool IsTransient(BaseEntity obj)

{

return obj != null && Equals(obj.Id, default(int));

}

private Type GetUnproxiedType()

{

return GetType();

}

public virtual bool Equals(BaseEntity other)

{

if (other == null)

return false;

if (ReferenceEquals(this, other))

return true;

if (!IsTransient(this) &&

!IsTransient(other) &&

Equals(Id, other.Id))

{

var otherType = other.GetUnproxiedType();

var thisType = GetUnproxiedType();

return thisType.IsAssignableFrom(otherType) ||

otherType.IsAssignableFrom(thisType);

}

return false;

}

public override int GetHashCode()

{

if (Equals(Id, default(int)))

return base.GetHashCode();

return Id.GetHashCode();

}

public static bool operator ==(BaseEntity x, BaseEntity y)

{

return Equals(x, y);

}

public static bool operator !=(BaseEntity x, BaseEntity y)

{

return !(x == y);

}

}

}



